Notes from the Book *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*

Babacar Niang

August 20, 2024

1 Practical Machine Learning Project

The goal of this chapter is to implement an algorithm to predict the median housing price of any district in California given data from a previous year. The approach will involve a supervised learning algorithm in batch learning, as there is no real-time data to be added, so an online learning method is not recommended. For the performance measure of our algorithm, we are going to use the **Root Mean Square Error** (RMSE), which is the most commonly used measure for regression problems. The algorithm is described as follows:

$$\text{RMSE}(\mathbf{X}, h) = \sqrt{\frac{1}{m}\sum_{i=1}^{m}\left(h\left(\mathbf{x}^{(i)}\right) - y^{(i)}\right)^2} \tag{1}$$

- m: The number of instances in the validation set.
- $\mathbf{x}$: The vector of all the features in a given instance i.
- y: The label of instance i, meaning its desired output.
- $\mathbf{X}$: The matrix containing the feature values of all the instances in the validation set.
- h: Your hypothesis (the prediction algorithm).

First, we will discuss downloading the data, cleaning it up, adding the necessary features, and finally, we will select a model and train it.

1.1 Download and Prepare the Data

In a real-life project, fetching the data would require querying a database, or going through spreadsheets and getting familiar with the data format. But for us, we will just have to download it from the GitHub link and unzip it in our current directory. To do that, it is always recommended to write a function that will automate the process, as the data is likely to be updated. In our code, we refer to it as the *load_data()* function. When called the function looks for the `datasets/housing` directory if it does not exist it creates it and donwload housing.tgz inside of it. After that it extracts the content of the donwloaded file and convert the data into a pandas Dataframe. Firstly it is always a good idea to check the content of the data, for this purpose the pandas dataframe function *head()* display by default the first 5 rows of the dataframe. We can observe that the data is composed of district (one per row) and each row possess 10 attributes per row. Moreover we can get a description of the data using the dataframe function *info()*; it will display the attributes, the number of rows in the data, the type of each row ...etc. Going throught the info we can observe that the ocean_proximity attribute is an object instead of a numerical one knowing that machine learning algorithms generally are sensible to that we will need to handle that case; we can also see that the values of this attribute are repetitive so it is also a categorical attributes. To display the values of the attribute, we can use the *value_counts()* function on the ocean_proximity column.

Now to look at the numerical attributes we can use the *describe()*. A point to note is the rows 25%, 50%, 75% shows the percentile of entries that fall under the displayed values. Additionally to the function outlined we can make a plot of the data to gain insight of it (as we did in the beginning of our *main()* function).
Important Note: It is very important to learn how the data was computed as this is deterministic in our model choice later. For example, in our data the median_income is capped to around 15.0 which is a scale so 15.0 means 150,000$.
Adding to the capping of the data we can observe that some other attributes are also capped (the median_house_value and the house_median_age), we can also see that the scale of the attributes are different from one another we will to mitigate that as well and finally the plots shows that most attributes are skewed-right (meaning they have a tendency of having more data point at the right of their center than on the left of it).

1.2 Creation of a test set

To validate our model at the end of the development phase, we need to separate our dataset into a test set and a training set so that we will avoid using an algorithm that overfits the data. Although it is possible to sample the dataset randomly when it is large enough, it is rather a better idea to stratify (divide the dataset into mutiple homogenous groups) our dataset so that the test set is representative of the real data. Since we have assumed that the median income is very important for the prediction of the median house value so we want to make sure that all the categories of income are represented in each stratum (a split of the dataset). In our code we first created the income category column (line 66) after that we begin to stratify the data using *sklearn.model_selection.StratifiedShuffledSplit()* and pass it as argument the number of strats we want to create. For this project we will only make one and for that a shorter way is to use the *sklearn.model_selection.train_test_split* which create one stat with adding the housing income category as the determinor for the splitting of the dataset. Now for further exploration we will create other attributes that will combinations of the existing ones. In our case we have added the bedroom_ratio, rooms_per_house and people_per_house attributes. Now we can pass to the prepraration of the data.

1.3 Preparation of the data

Preparing the data has for function to make our dataset more suitable (removing the rows with missing features, scaling the data properly ...etc) for the training phase. First we will have to separate our data from the labels (the label is what our model should be able to predict in our case it is the housing median value). Now let's clean up the rows that miss features, the options we have is to remove those rows entirely, to get rid of the attribute(s) or to fill those missing values with something (the mean, the median or the most recurring value). We will go for the third option, for that we will use the *SimpleImputer* class from sklearn.imputer. Notice that in our code we created a dataframe with only the numerical attributes first before applying the imputer to it (as the imputer does not work on non-numerical attributes). We first train the imputer to calculed the median of each numerical attributes (line 93) and we finally we used the imputer to transform the numerical attributes (by calling the transform function

on the imputer and passing it the housing_num dataframe line 95). Note that we can also replace the missing values with the mean or the most frequent value by changing the strategy argument on the imputer class. After that we need to handle categorical attributes, the most common solution is to encode this attributes and transform them in numbers. In order to do that we have at our disposal the *OrdinalEncoder()* from sklearn.preprocessing. But this method of encoding may not be optimal so instead we should be using the *OneHotEncoder()* which uses a scipy sparse matrix to store the state of each category per row, the state being a binary value (for example if a district is near the ocean his near_ocean value will be one and the inland and all the other categories for him will be 0). The implementation demonstrate that at line 98. And now we need to take care of the different scaling of the attributes. For us it is better to use the *StandardScaler()* class from the sklearn.transformer. The problem with using a standard scaler is that it break whenever the distribution of the attribute isn't even. To solve that problem we can bucketize the distribution into buckets of equal size. The most common way of doing it involve using the **RBF** (radial basis function) which measure the distance between the inputed value and a fixed point (in our case the mean). As we can observer our implementation uses the gaussian rbf (line 106) to calculates the similarity between the housing median age attribute of all the houses in the dataset and a specific value of 35 years.